Supporting Software-Evolution at the Process Level

Tilman Seifert and Markus Pizka

Technische Universität München, Germany
Software & Systems Engineering
{seifert,pizka}@in.tum.de

Abstract. The ability to be changeable is inherent to software — in fact, this is what defines it as being "soft". The long-term management of large software systems depends on the ability of a system to be "easy" to maintain and evolve. In contrast to commonly presented views, we define three ways to look at evolvability. First, it can be considered to be a quality property, and must therefore be subject to quality control. Second, it can be handled as a non-functional requirement and must therefore be part of the requirements management and change management process. Third, one could claim that it is not the system's ability but the ability of the development and maintenance team that allows for cost-effective maintenance cycles. This in turn requires a sound understanding of the system and its environment. This paper argues that the combination of these three different views allows for a coherent understanding of maintenance and evolution of software systems. We suggest a couple of principles how to deal with long-lived systems in a systematic fashion, and derive a development process model built on these principles.

1 Introduction

In [HT03], Hunt and Thomas argue that "there's no substitute for experience" in judging design decisions. This paper contrasts this opinion: Experience of course is among the most valuable assets. But experience is the sum of lessons learned, about successful as well as failed projects, approaches, and ideas. For successful projects in an ever-changing environment, experience must be combined with creativity and the courage to try new ways, and new experiences come only from new ideas. Lessons learned by experienced people have to be passed on to junior folks — if that isn't done, it's just poor knowledge management.

Hunt and Thomas propose a balance between YAGNI ("you ain't gonna need it", as promoted by Kent Beck and the XP community [Bec99]) and DOGBITE ("do it, or get bitten in the end", suggested by Hunt/Thomas), and they propose to find that balance by experience. While we strongly support this balance, we argue for a systematic approach to reach it. The "Pizka-Seifert-Process Model" which we introduce in section 4 provides the needed ingredients to achieve this goal.

Maintainability and evolvability are quality aspects of any system, and they are non-functional requirements for all systems. To achieve quality in general and maintainability and evolvability in particular, there are several approaches:

The Organizational Way: This is about proper project management and appropriate development process models. It also includes team management and knowledge management.

The Technical Way: There are again several approaches:

– Use and reuse standard components (like e. g. Meyer suggests [Mey03]).
– Use Standard Architectures: Reuse not code but ideas. This is basically a pattern approach, using "best practices".
– Model-Based Development: Concentrate on the new aspects of your system, and leave other issues to the CASE tool. The aim is to formulate concepts on a higher level of abstraction.

This distinction can also be found in [Mey03]. These approaches are non-exclusive and need to be combined effectively. We introduce a software process model that integrates well-known techniques like incremental development or architecture refactoring with a systematical approach that explicitly alternates between active development stages and consolidation stages. This process model is based on the observation that on one hand the technical foundation of a system might not be prepared for certain changes and must therefore be changed from time to time, but on the other hand the system's growth needs to be planned carefully, or otherwise it will be difficult to handle fairly soon.

This article is organized as follows: Section 2 introduces three views on the evolvability of a software system. Section 3 names principles for long-term management of large software systems. Section 4 introduces a new process model for the management of long-lived software systems and contrasts it to the maintenance practices of the SPiCE model [SPI95].

2 Three Ways to See Evolvability

The process of evolution and maintenance is very complex — that's why many authors seem to think of experience as the only way to deal with it. We believe that it is now time to de-mystify this area, and propose a systematic approach to analyze important factors, and derive a process model to deal with this complexity in a systematic fashion.

2.1 Evolvability as a Quality Property

A software system, or a part thereof, like a component, might be put into a different context than the one which it was originally developed for. Reasons are manifold: functional and non-functional requirements evolve, the system might need to scale in a different way, it could be reused in a different environment, or neighboring systems might change. The adaptability of the software to the new context can be expressed by quantifying the effort needed to make it work correctly and useful. This adaptability can be regarded as one quality property of the software.

Therefore it seems obvious that maintainability and evolvability are subject to the quality management process. But this in turn implies that the QA process must be capable to consider and control the needs for evolvability. For this, let us consider the exact properties that define evolvability. Only a system in which every part is understood can possibly be maintained. This requires a technically sound architecture and capable developers. A somehow complete and accurate (we didn't say "lengthy") up-to-date documentation is typically very helpful.

It is difficult enough to develop an architecture that is at the same time simple and easy to understand, precisely defined, and powerful enough to express all important concepts of the system. Concerning design policies for good architectures, there is a long list of literature describing principles for good design of interfaces, components, or architectures, covering at least three decades of research on Software Engineering, e. g.: Parnas, 1972: Separations of Concerns [Par72], Meyer, 1997: OO Design and Design by Contract [Mey97], Szyperski, 1998: Component Software, [Szy98], Beck, 1999: YAGNI [Bec99], or Bass et al., 2003: Software Architecture [BCK03].

Yet, it is even more difficult to develop an architecture with all the properties mentioned above and the additional ability to be flexible and adaptable. No matter how flexible an architecture is designed, in a changing environment architectures will have to change, too. This is not a bad thing in itself, but architectural changes have to be accomplished within an appropriate process.

2.2 Evolvability as a Non-Functional Requirement

Evolvability and maintainability can also be regarded as non-functional requirements. They are defined by the expectations for the lifetime, changing environments, or changing functional requirements. This view has some interesting implications:

– The wish for "easy maintainability" needs to be expressed explicitly, and the conditions under which maintenance or evolvements might happen, need to be qualified.
– Different scenarios of possible system evolution will be set up.
– Being part of the requirements, the trade-off between time-to-system and maintainability will explicitly be dealt with by giving priorities to requirements. By introducing expectations about possible changes to the system, the decision about the needed "elegance" of a solution is now shifted from the programmer or designer to the customer. This is a chance to involve the customer into technical decisions in terms that he or she can actually relate to.

Since these expectations are by their nature subject to uncertainty, they should be dealt with in the risk management process.

2.3 Evolvability as an Ability of the Team

The properties mentioned above, that define the effort needed to maintain or evolve a software system, are assumed to be relative to the knowledge and experience a developer or a development team has with this particular system or with systems of this particular kind.

This is true to some extent, but in the light of the long periods of time that many systems are in productive use, the fluctuation of developers, and the aim to improve the accuracy of planning, there is the wish to decouple the term "maintainability" of a system from the persons actually working on the system. This reasoning is the foundation for postulating concise documentation and an architecture that is relatively easy to grasp. At the same time, we are proposing an approach that is based on a systematic reasoning instead of just relying on the experience of people as long as they are available.

3 Guiding Principles of System Evolution

We consider some general principles as essential for successful long-term management of large software systems.

1. Tiny changes to the way the system is used entail minor changes to the system itself, and eventually its quality deteriorates. A redesign of the system is neither necessary nor desired for every tiny change—but it must not be forgotten. An evolution process must therefore allow for both small changes and major revisions.
2. Continuous consolidation: In the long run, the system integrity must be kept. Once it is lost, it is hard or impossible to regain.
3. Working knowledge about the system is required and must be kept. Once it is lost, it may be hard to regain.
4. The economic side: Long-lived software systems are expensive. Business goals must be clearly defined in order to find the appropriate technical solution.
5. For any project it is common sense that all development steps have to be planned. Let's call these steps "micro steps"—and it becomes evident that the "macro steps" which define the evolutionary changes of a software systems, need to planned very carefully.

4 The Pizka-Seifert Process Model

When contrasting the YAGNI approach and the DOGBITE statement, we feel reminded on the discussion of "bottom-up" vs. "top-down" approaches to requirements and implementation. Both approaches have their advantages and disadvantages.

Strict bottom-up development would mean trying to reach the requirements by creating higher level services based on already known or available concepts and components. By this, bottom-up development proceeds rapidly and delivers highly efficient systems that might well miss their actual requirements. Top-down development in contrast, means to start with the requirements specification and refine them stepwise down to the technical infrastructure. Trivially, the resulting system will perfectly match its requirements but it may well miss technical reality, which in turn must be compensated with complicated and inefficient mappings to the execution environment.

In order to benefit from the advantages of both sides and eliminate their shortcomings, a development process needs to combine both into one process. As we will show below, this can be achieved by alternating between evolution and consolidation phases. This would be an enhancement of the "Staged Software Life Cycle Model" as proposed by Rajlich et al. [RB00].

4.1 The Staged Software Life Cycle Model

In [RB00], Rajlich and Bennett introduce the “staged software life cycle model”: According to this model, the life cycle of a software system starts with *initial development* where a first fully functional version of the software is produced. The system moves on to *evolution stage*, during which the system’s functionality is enhanced or adopted in order to better serve the users’ needs. The next phase is the *servicing phase* which only allows for minor repairs and small functional changes. From there, it is inevitable that the system eventually passes on to the *phase-out* stage where the system is being kept alive but is not changed any more. Typically, this is due to the fact that no developer or maintainer dares to touch the system any more. Finally, the system is *closed down* and replaced by it successor.

4.2 Critique

This model is intuitively helpful for the description of long-lived systems. It certainly helps e. g. in discussions between management and technical staff about the state of a system and necessary technical decisions and their consequences.

However, it stays rather vague on issues that would be important for constructive steering of system evolution. The model does not give any hints on how to stay in the evolution stage as long as possible. It uses but does not define the term “architectural integrity” that — according to the model — seems to be one of the major pillars on which evolvability relies.

It furthermore states too rigidly that systems can not return from servicing back into evolution. There are several counterexamples to this, if we think for example of Open Source Software such as Linux or commercial products, such as SAP, that were successfully serviced and evolved in several iterations over long periods of time.

In addition to this, we believe that the initial development should not be viewed separately from the rest of the life cycle. First, it has a decisive impact on the life-time of the system. Second, long running initial developments are themselves composed out of evolution and servicing steps.

4.3 The PSPM Approach

With the PSPM (Pizka-Seifert Process Model) we define a model that can be used for constructive planning of a system’s life cycle. The PSPM takes software evolution from mere descriptions of a system’s state to actual planning steps including both management and technical decisions. We base our model on the three views on system evolution introduced in section 2 and adhere to the guiding principles given in section 3.

Figure 1 illustrates the PSPM. From the very beginning of initial development the system enters a process that alternates between evolution and consolidation phases. This process spans the complete life cycle of the system until its phase out. Between the major phases evolution and consolidation, the system is serviced, that is minor corrections or enhancements are performed. According to number 1 of our guiding principles a single servicing activity does not degrade the quality of the system enough to necessitate

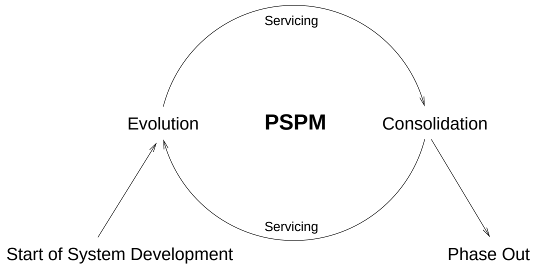

Fig. 1. PSPM Software Life Cycle

major rework. But increasing numbers of tiny changes together with their interplay may cause serious risks in the long run.

The accumulation effect is encountered with an explicit consolidation phase, which we will explain in more detail below. The consolidation phase constitutes the bottom-up portion of the process. The existing system is taken and modified according to technical aspects without actually adding new features but changing what is already there.

The evolution phase is the top-down part of the PSPM. In this phase, requirements are elicited, refined and corresponding features are integrated into the system. The primary focus of this phase is to implement the requirements and not the elegance of the technical solution.

We state that both phases are essential and must be carried out in systems development one way or the other. The PSPM differs from other iterative processes models significantly by respecting the role of both activities at the process level. By this, evolution becomes more controllable, consolidation steps are pre-planned, the development team can be organized accordingly, and planned consolidation activities can be communicated to the customer before they are unavoidable.

4.4 Software Consolidation

It is important to actively avoid serious troubles *before* they happen. For this, we have to “clean up” the system from time to time. We call this clean up consolidation. It can be viewed as “preventive maintenance”.

During this phase, no new features are implemented. Instead, we check the consistency of documentation, we look for concepts that are distributed over the system and could be centralized, components that grew too large and should now be split, components that interact heavily or inefficiently, dead code, redundancies and much more. Briefly, the system is refactored into a consistent state and freed from redundancy. As consolidation is very different from the development of new concepts it needs to be carried out by experts with particular skills and techniques.

Architecture consolidation itself is not completely new but happens in many maintenance projects. But usually, it is applied *after* serious problems have appeared. At this time, the system is often already in a state that is very difficult to comprehend or change. Consolidation becomes complicated, costly, risky and takes too long. Thus, an assessment of the usefulness of the system might indicate the replacement of the system

[STS97] although it is well-known, that large scale software development projects carry high risks [CHA99].

We state that this undesirable situation can be avoided or at least significantly delayed. We therefore institutionalize regular consolidation as an important and productive part of the life cycle of the system. This way, consolidation comes in as a preventive measure instead of a counter-measure against tough problems. Our aim is to keep the system in a “healthy” state and to maintain it as clear and concise as possible.

4.5 Discussion

Working knowledge about the system is very important — but it is as important to be open and to let in new expertise. The alternation between development and consolidation assigns a high value to working knowledge and experience, and it also embraces new views on the system.

Technical decisions always have a price, and some changes to software systems involve serious investments. Therefore, it is necessary to take business decisions about how much to invest in a software system consciously. The systematic, a priori planned consolidation phases align technical decisions with business goals. We introduce the expectations about the lifetime of the system explicitly as a relevant factor for design decisions and investment decisions. Thereby, we can avoid the classic dispute about “nice” and “cheap” solutions, typically carried out between technical staff and their managers.

In terms of the Staged Software Life Cycle Model, our model allows for keeping large software systems in the “evolution stage” as long as wanted in an actively planned way.

It is useful to compare our new process model with a well-known capability model in order to detect strengths and weaknesses. The SPiCE model [SPI95] for example considers several engineering practices. Concerning “maintenance” it proposes that the process should “respond to” changing environments, which would be a *reactive* process. What is actually needed are *active* processes that do not only perform change management, but support active steering of the software evolution. The PSPM does support long-term management of system evolution that goes beyond the planning of single projects and the mere reaction to change requests.

5 Conclusion

The PSPM model paves the ground for new constructive ways to manage large software systems over long periods of time. Technical questions are aligned with business goals.

A currently ongoing consolidation project in a large, long-lived information system shows very promising results supporting the PSPM. Our approach achieves a high level of satisfaction among developers due to its constructive nature and the four-eye-principle which brings in new expertise in addition to the available experience with the system.

Of course, open issues remain. For brevity, this paper only considers the process aspects. Other details like organization or technical aspects need to be discussed in greater depth. Some questions need more research, such as advanced metrics or heuristics that indicate when to switch from evolution to consolidation.

Acknowledgments

This work was funded by the research project “ViSEK”, granted by the German Department of Education and Research, BMBF.

References

[BCK03] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison Wesley, 2003.

[Bec99] Kent Beck. *Extreme Programming Explained – Embrace Change*. Addison-Wesley, 1999.

[CHA99] CHAOS – a Recipe for Success. The Standish Group, 1999.

[HT03] Andy Hunt and Dave Thomas. The Trip-Packing Dilemma. *IEEE Software*, 20(3):106–107, May 2003.

[Mey97] Bertrand Meyer. *Object-oriented Software Construction*. Sams, 1997.

[Mey03] Bertrand Meyer. The Grand Challenge of Trusted Components. In *25th International Conference on Software Engineering*, pages 660–667. IEEE Computer Society, May 2003.

[Par72] David L. Parnas. On the Criteria to be used in Decomposing Systems into Modules. *CACM*, 15(12):1053–1058, 1972.

[RB00] Václav T. Rajlich and Keith H. Bennett. A Staged Model for the Software Life Cycle. *IEEE Software*, 33(7):66–71, July 2000.

[SPI95] The SPiCE Project – Software Process Improvement and Capability Determination. Technical report, ISO/IEC, 1995.

[STS97] STSC. Software Reengineering Assessment Handbook v3.0. Technical report, STSC, U.S. Department of Defense, March 1997.

[Szy98] Clemens Szyperski. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1998.